Some Autonomic Properties of Two Legacy Multi-Agent NASA Systems - LOGOS and ACT

Walt Truszkowski, James Rash
NASA GSFC
Code 588
walter.f.truszkowski@nasa.gov
james.l.rash@nasa.gov

Christopher Rouff
SAIC
rouffc@saic.com

Mike Hinchey
NASA GSFC
Code 581
michael.g.hinchey@nasa.gov

Abstract

To reduce the cost of future space flight missions and to perform new science, NASA has been investigating autonomous ground and space flight systems. These goals of cost reduction have been further complicated by NASA's plans to use constellations and swarms of nanosatellites for future science data-gathering which may entail large communications delays and loss of contact with ground control for extended periods of time. This paper describes two prototype agent-based systems, the Lights-out Ground Operations System (LOGOS) and the Agent Concept Testbed (ACT), and their autonomic properties that were developed at NASA Goddard Space Flight Center (GSFC) to demonstrate autonomous operations of future space flight missions. The paper discusses the architecture of the two agent-based systems, operational scenarios of both, and the two systems' autonomic properties.

1. Introduction

Until recently, space missions have been operated manually from ground control centers. The high costs of satellite operations have prompted NASA and others to seriously look into automating as many functions as possible. A number of more-or-less automated ground systems exist today, but work continues with the goal of reducing operation costs to even lower levels. Cost reductions can be achieved in a number of areas. Greater autonomy of satellite ground control is one such area.

To accomplish these cost reductions, NASA has set far-reaching autonomy goals for ground-based and space-based systems. More reliance on "intelligent" systems and less on human intervention characterizes its autonomy goals. These goals of cost reduction have been further complicated by NASA's plans to use constellations and swarms of nanosatellites for future science data-gathering. These constellations may have large communications delays and at times be out of contact with ground control for extended periods of time. Because of this, the onboard and ground-based systems will need to have autonomic properties. The remainder of this paper discusses two agent-based systems that have been prototyped and their autonomic properties.

2. Overview of Two NASA GSFC Agent-based Systems

The Advanced Architectures and Automation Branch at Goddard Space Flight Center (GSFC) has a leading role in the development of agent-based approaches to realize NASA's autonomy goals. Two major successes of the branch were the development of the Lights-Out Ground Operations System (LOGOS) [3, 4, 6, 7] and the Agent Concept Testbed (ACT) [6, 7].

LOGOS was the first multi-agent system developed and provided an initial insight into the power of agent communities, autonomy and autonomic properties of these systems. The agents in LOGOS acted as surrogate controllers and interfaced with legacy software that controllers normally used as well as humans.

Based on the success of this first prototype, development was done on ACT, an environment in which richer agent and agent-community concepts were developed through detailed prototypes and operational ground-based and space-based scenarios. ACT has given GSFC more experience in architecting and developing communities of agents, autonomous and autonomic systems, as well as the trade-offs and complications that accompany such systems.

The goal of our work is to transition proven agent technology into operational NASA systems. The implementation of the above multi-agent systems provided an opportunity to exercise and evaluate the capabilities supported by the agent architectures and refine the architectures as required. It also provided an opportunity for space mission designers and developers to "see" agent technology in action. This has enabled them to make a better determination of the role that agent technology can play in their missions.

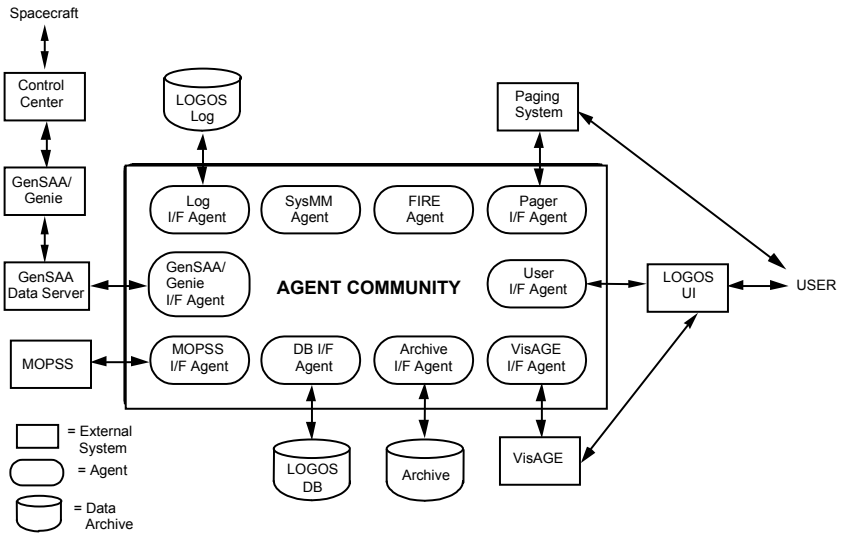

Figure 1: LOGOS agent community and legacy software.

There have been many definitions of agents [1, 8]. For this paper we will define an agent as a software system that is autonomous and has the ability to perceive and affect its environment and communicate with other agents. A multi-agent system, or community of agents, is simply a collection of agents that cooperate in a community to accomplish a common goal. For autonomic systems we will use the definitions provided in [9], self-configuring, self-optimizing, self-healing, and self-protecting. The remainder of this paper describes the LOGOS and ACT agent communities, brief operational overviews and the autonomic properties of each of the two systems.

3. LOGOS

LOGOS was a proof-of-concept system that used a community of autonomous software agents that cooperated to perform functions previously done by human operators who used traditional software tools, such as orbit generators and command sequence planners. The following discusses the LOGOS architecture, an operational scenario and its autonomic properties.

3.1. LOGOS Architecture.

The LOGOS architecture is shown in Figure 1. LOGOS was made up of 10 agents, some of which interfaced with legacy software, some which performed services for the other agents in the community, and others which interfaced with an analyst or operator. All agents could communicate with all other agents in the community, though not all agents needed to communicate with each other.

The System Monitoring and Management Agent (SysMMA) kept track of all agents in the community and provided addresses of agents for other agents requesting services. Each agent when started had to register their capabilities with SysMMA and obtain addresses of other agents whose services they would need.

The Fault Isolation and Resolution Expert (FIRE) agent resolved satellite anomalies. FIRE was notified of anomalies during a satellite pass. FIRE contained a knowledge base of potential anomalies and a set of possible fixes for each. If it did not recognize an anomaly or was unable to resolve it, it sent the anomaly to the user interface agent to be forwarded to an analyst for resolution.

The User Interface Agent (UIFA) was the interface between the agent community and the graphical user interface that the analyst or operator used to interact with the LOGOS agent community. UIFA received notification of anomalies from the FIRE agent, handled the logon of users to the system, kept the user informed with reports, and routed commands to be sent to the satellite and other maintenance functions. If the attention of an analyst was needed but none was logged on, UIFA would send a request to the PAGER agent to page the required analyst.

The VisAGE Interface Agent (VIFA) interfaced with the VisAGE 2000 data visualization system. VisAGE was used to display spacecraft telemetry and agent log information. Real time telemetry information was displayed by VisAGE as it was downloaded during a pass. VIFA requested the data from the GIFA and AIFA agents (see below). An analyst could also use VisAGE to visualize historical information to help monitor spacecraft

health or to determine solutions to anomalies or other potential spacecraft problems.

The Pager Interface Agent (PAGER) was the agent community interface to the analysts' pager system. If an anomaly occurred or other situation arouse that needed an analyst's attention, a request was sent to the PAGER agent which then sent a page to the analyst.

The Database Interface Agent (DBIFA) and the Archive Interface Agent (AIFA) stored short term and long term data, respectively, and the Log agent (LOG) stored agent logging data for debugging, illustration and monitoring purposes. The DBIFA stored such information as the valid users and their passwords and the AIFA stored telemetry data.

The GenSAA/Genie Interface Agent (GIFA) interfaced with the GenSAA/Genie ground station software, which handled communications with the spacecraft. GIFA had the capability to download telemetry data, maintain scheduling information and to upload commands to the spacecraft. Upon downloading anomalies and other data from the spacecraft, GIFA routed the data to other agents based on their requests for information.

The MOPSS (Mission Operations Planning and Scheduling System) Interface Agent (MIFA) interfaced with the MOPSS ground station planning and scheduling software. MOPSS kept track of the satellite's orbit and the time of the next pass and how long it would last. It also sent out updates to the satellite's schedule to requesting agents when the schedule changed.

3.2. An example Scenario

An example scenario of how agents in LOGOS communicated and cooperated would start with MIFA receiving data from the MOPSS scheduling software that the spacecraft would be in contact position in two minutes. MIFA would then send a message to the other agents informing them of the upcoming event in case they needed to do some preprocessing before the contact. When GIFA received the message from MIFA it sent a message to the GenSSA Data Server to start receiving transmissions from the control center.

After receiving data, the GenSSA Data Server sent the satellite data to GIFA, which had rules indicating what data to send to which agents. As well as sending data to other agents, GIFA also sent all engineering data to the archive agent (AIFA) for storage and trend information to the visualization agent (VIFA). GIFA sent updated schedule information to the scheduling agent (MIFA) and sent a report to the user interface agent (UIFA) to be sent on to an analyst for monitoring purposes. If there were any anomalies, GIFA would send them to the FIRE agent for resolution.

If there was an anomaly, the FIRE agent would try to fix it automatically by using a knowledge base containing possible anomalies and a set of possible resolutions for each anomaly. To fix an anomaly, FIRE would send a spacecraft command to GIFA to be forwarded on to the spacecraft. After exhausting its knowledge base, if FIRE was not able to fix the anomaly, then FIRE forwarded the anomaly to the user interface agent, which then paged an analyst and displayed it on their computer for action. The analyst would then formulate a set of commands to send to the spacecraft to resolve the situation. The FIRE agent upon receiving the commands would add the new resolution to its knowledge base for future reference and would send the commands to the GIFA agent, which would send them to the GenSAA/Genie system for forwarding on to the spacecraft.

There were many other interactions going on between the agents and the legacy software, which was not covered above. Examples include the DBIFA requesting user logon information from the database; the AIFA requesting archived telemetry information from the archive database to be sent to the visualization agent and the pager agent sending paging information to the paging system to alert an analyst of an anomaly needing his or her attention.

3.3. Autonomic properties of LOGOS.

The operational scenarios of LOGOS exhibit the four types of autonomic properties: self-configuring, self-optimizing, self-healing and self-protection. The following discusses each of these properties of LOGOS in more detail.

3.3.1. LOGOS self-configuring. LOGOS self configures when the GIFA agent gets signals from the GenSAA/Genie ground station software that a spacecraft pass is about to happen. When this occurs, the GIFA configures the system by waking up the needed agents for the pass. If there are no anomalies, then the FIRE agent is not needed and is not woken up. If it is needed, then LOGOS is configured for that pass with the FIRE agent woken up and ready to receive the anomaly. The same is true of the visualization agent and the user interface agent if there is no user currently logged on, then those agents do not have to be woken up for the spacecraft pass.

3.3.2. LOGOS self-optimizing. LOGOS self-optimizes itself through learning. One example of this is through the learning that the FIRE agent does when it does not know how to fix an anomaly and notifies an analyst that it needs help. After the analyst provides a set of commands to fix the anomaly, the FIRE agent stores those commands and the parameters to that anomaly in its knowledge base for

future reference. In this way it can fix this future problem when it occurs again.

A second way that LOGOS self optimizes is through the user interface agent and the visualization agent. These agents stored information that specifies which analyst looks at what data so that that information would be pre-fetched and available to the analyst when he or she logs on to the system. This saves time for the analyst, especially in a critical situation, so anomalies or just their normal job in general can be done faster.

A third example is the pager agent, which notifies analysts when an anomaly is present. This agent also kept track of information that specified which analysts were available at what times and modified who it called first based on their usual availability.

3.3.3. LOGOS self-healing. LOGOS self-heals primarily through the actions of the FIRE agent. In this case, the FIRE agent examines anomalies that occur and then issues commands to fix/heal the anomalies based on its knowledge base. It also self-heals through the intervention of the human in the loop who can fill in information when the FIRE agent does not have the requisite knowledge to solve a problem. In this case the human is viewed as part of the overall system architecture. The FIRE agent would also learn how to fix future anomalies based on the inputs from the analyst when the FIRE agent needed help. The self-healing aspect of LOGOS was its primary function and is what made the system lights-out and enabled lower costs of future operations through reduced man-power requirements.

3.3.4. LOGOS self-protecting. The self-protecting aspects of LOGOS are limited. The self-protection is primarily done by the FIRE and the user interface agent. For the user interface agent, self-protection is accomplished when it authenticates a user logging on to the system to ensure the user has proper credentials before allowing him/her to send commands to the spacecraft. For the FIRE agent, self-protection is accomplished when checking commands entered by the analyst to ensure they are not going to harm the spacecraft, though this could be overridden. By checking the commands before sending them to the spacecraft, possible errors in the analysts commands can be found and possible damage to the spacecraft can be adverted.

4. ACT

The ACT agent architecture provides for a flexible implementation of a wide range of intelligent or reactive agents. Agents in ACT are built using a component architecture where a component can be easily swapped out and replaced by another more advanced component. The component architecture allows for easy removal of unneeded components for reactive agents and the inclusion of the necessary components to implement intelligent agents. It is also flexible so that additional unforeseen components implemented with new AI technologies can be added as they become available without affecting previously implemented components.

4.1. The ACT Architecture

The ACT component-based architecture allows greater flexibility to agent designers. A simple agent can be designed by using a minimum number of components that receive percepts (inputs) from the environment and react according to those percepts. This type of agent would be a reactive agent.

A robust agent would be designed using more complex components that allow the agent to reason in a deliberative, reflexive and/or social fashion. This robust agent would maintain models of itself, other agents in its environment, objects in the environment that pertain to its domain of interest, and external resources that it might utilize in accomplishing a goal. Figure 2 depicts the components for a robust ACT agent. These components give the agent a higher degree of intelligence when interacting with its environment.

The following sections describe the components listed in Figure 2 and the framework in which the components are implemented.

4.1.1. Modeler. The modeling component is responsible for maintaining the domain model of an agent, which includes models of the environment, other agents, and the agent itself. The Modeler receives data from the Perceptors and agent communication component. This data is used to update state information in its model. If the data causes a change to a state variable, the Modeler publishes this information to other components that have subscribed to updates to that variable. The Modeler is also responsible for reasoning with the models to act proactively and reactively with the environment and events that affect the model's state. In the future the Modeler will also dynamically modify its model based on experience.

The modeler can also handle what-if questions. These questions would primarily come from the planning and scheduling component, but may also come from other agents or from a person who wants to know what the agent would do in a given situation or how a change in the agent's environment would affect the values in its model.

4.1.2. Reasoner. The Reasoner component works with information in its local knowledge base as well as model and state information from the Modeler to make decisions

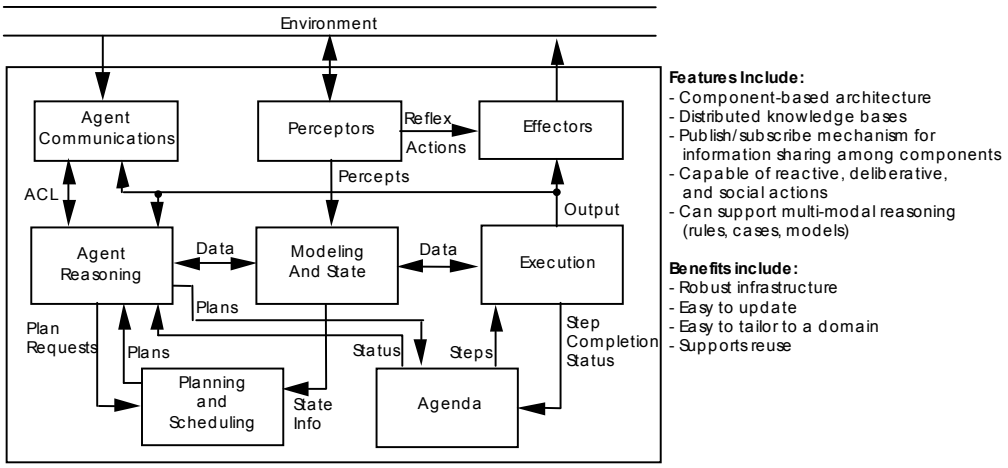

Figure 2: ACT Agent Architecture

and formulate goals for the agent. This component reasons with state and model data to determine whether any actions need to be performed to affect the agent's environment, change its state, perform housekeeping tasks, or influence other general activities. The Reasoner will also interpret and reason with agent-to-agent messages. When action is necessary for the agent, the Reasoner will produce goals for the agent to achieve. Currently the Reasoner works more in a reactive manner. Either an input coming in or a trigger from the clock sets it in motion. Work is also being done to make the Reasoner more proactive.

4.1.3. Planner/scheduler. The Planner/Scheduler component is responsible for any agent level planning and scheduling. The planning component receives a goal or set of goals to fulfill in the form of a plan request. This typically comes from the Reasoning component, but may be generated by any component in the system.

At the time a plan request is received, the planning and scheduling component acquires a state of the agent and system, usually the current state, as well as actions that can be performed by this agent (typically from the modeling and state component). The planning and scheduling component then generates a plan as a directed graph of steps, which is composed of preconditions to check, an action to perform, and expected results (post condition). Each step is also passed to any Domain Expert components/objects for verification of correctness. If a step is deemed incorrect or dangerous, the Domain Expert may provide an alternative step or solution to be considered by the planner. Once the plan is completed, the Planner/Scheduler sends it back to the component that requested the plan (usually the Reasoner). The requesting component then either passes it on to the Agenda to be executed or uses it for planning/what-if purposes.

4.1.4. Agenda/executive: The Agenda and Executive work together to execute the plans developed by the Planner/Scheduler. The agenda typically receives a plan from the Reasoner, though it can receive a plan from another component that is acting in a reactive mode. The agenda interacts with the Execution component to send the plan's steps, in order, for execution. The agenda keeps track of which steps are being executed, finished executing, idle, or waiting for execution. It updates the status of each step appropriately as the step moves through the execution cycle. The agenda reports the plan's final completion status to the Planner and Agent Reasoner when the plan is complete.

The Executive executes the steps it receives from the Agenda. If the preconditions are met, the action is executed. When executions finish, the Executive evaluates post-conditions, and generates a completion status for that step. The completion status is then returned to the agenda.

A watch, attached to the Executive, monitors given conditions during execution of a set of steps. Watches allow the planner to flag things that have to be looked out for during real-time execution, which can be used to provide "interrupt" capabilities within the plan. An example would be to monitor drift from a guide star while doing an observation. If the drift exceeds a threshold, then the observation is halted. In such a case the watch would notify the Executive which in turn would notify the Agenda. The Agenda would then inform the Reasoner that the plan failed and the goal was not achieved. The Reasoner would then formulate another goal (e.g., recalibrate the star tracker).

4.1.5. Agent communications: The agent communication component is responsible for sending and receiving messages to/from other agents. The component takes an agent data object that needs to be transmitted to

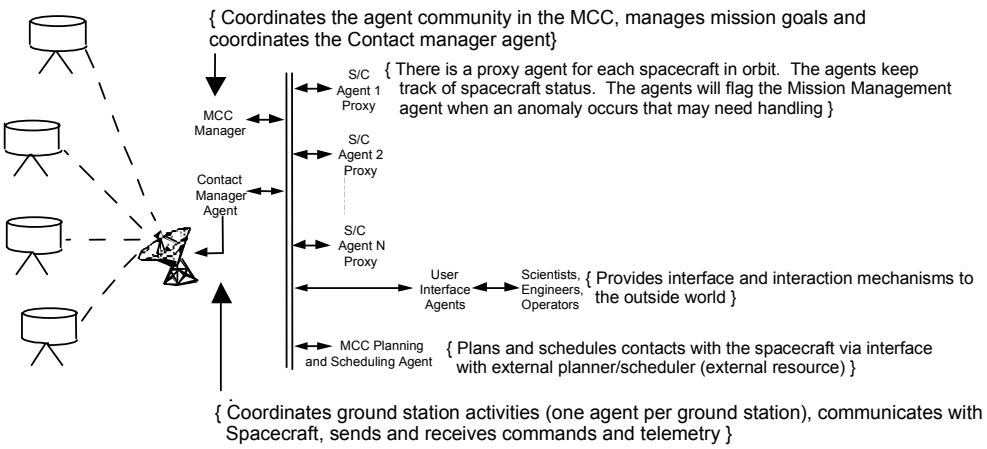

Figure 3: Agent community being developed in ACT to test out the new agent architecture and some community concepts.

another agent and converts it to a message format understandable by the receiving agent. The message format being used is based on Foundations of Intelligent Physical Agents (FIPA) [2]. The transmitting of a message to the appropriate agent occurs through the use of a NASA developed agent messaging software called Workplace [3].

The reverse process occurs for an incoming message. The communications component takes the message and converts it to an internal object and sends it to the other components that are subscribing to incoming messages. The communications component can also have reactive behavior where for a limited number of circumstances it produces an immediate response to a message.

4.1.6. Perceptors/effectors: The Perceptors are responsible for monitoring the environment for the agent. An example of an environment is a spacecraft subsystem. Any data received by the agent from the environment, other than agent-to-agent messages, enters through Perceptors. An agent may have zero or more Perceptors, where each Perceptor receives information from specific parts of the agent's environment. A Perceptor may just receive data and pass it on to another component in the agent or it may perform some simple filtering/conversion before passing it on. A Perceptor may also act intelligently through the use of reasoning systems if it is desired. If an agent is not monitoring the environment, then it would not have any perceptors (an example of this would be an agent that only provides expertise, such as fault resolution, to other agents).

The Effector is responsible for effecting or sending output to the agent's environment. Any agent output data, other than agent-to-agent messages, leaves through Effectors. Typically the data leaving the Effectors will be sent from the executive, which has just executed a command to send data to the environment. There may be zero or more Effectors, where each Effector sends data to specific parts of the agent's environment. An Effector may perform data conversions when necessary and may even act intelligently and in a proactive manner when necessary. As with the Perceptors, an agent may not have an Effector if it is not interacting with the environment.

4.1.7. Agent framework: A framework is used to provide base functionality for the components as well as the inter-component communication facility. The framework allows components to be easily added and removed from the agent while providing a standard communications interface and functionality across all components. This makes developing and adding new components easier and makes additions transparent to existing components in the agent. Each component in the architecture can communicate information to/from all other components as needed.

The primary communications for components is based on a publish-and-subscribe model with direct links between components when large amounts of data need to be transferred. Components communicate to each other the types of data that they produce when queried. When one component needs to be informed of new or changed data in another component, it subscribes to the particular data in the other component. Data can be subscribed to whenever it is changed or on an as needed basis. With this mechanism a component can be added or removed without having to modify the other components in the agent.

4.2. Dataflow between Components

This section gives an example of how data flows between components of the ACT architecture. The example being used is when a spacecraft's battery is discharging. The scenario reads as follows:

1. The agent detects a low voltage when reading data from the battery via a Perceptor. The Perceptor then passes the voltage value to the Modeler, which has subscribed to the Perceptor to receive all percepts.
2. When the Modeler receives the voltage from the Perceptor it updates the value in its model. In this case, the new value puts it below the normal threshold and changes the voltage state to "low". This causes a state change event and the Modeler publishes the new value to all subscribing components. Since the Reasoner is one of the subscribers, the low voltage data value is sent to the Reasoner.
3. In the Reasoner the low voltage value fires a rule in the expert system. This rule calls a method that sends the Planner/Scheduler a goal to achieve a battery voltage level that corresponds to a full charge.
4. When the Planner/Scheduler receives the goal from the Reasoner, it queries the Modeler for the current state of the satellite and a set of actions that can be performed.
5. After receiving the current state of the satellite and the set of available actions from the Modeler, the Planner/Scheduler formulates a list of actions that need to take place to charge the battery. It then sends the plan back to the Reasoner for validation.
6. The Reasoner examines the set of actions received from the Planner/Scheduler and decides that it is reasonable. The plans are then sent to the Agenda.
7. The Agenda then puts the action steps from the plan into a queue for the Executive.
8. When the Executive is ready to execute a new step, the agenda passes it one at a time for execution.
9. The Executive executes each action until the plan is finished and then notifies the Agenda it is done.
10. The Agenda marks the plan as finished and notifies the Reasoner that the plan finished successfully.
11. After the plan is executed, the voltage rises and triggers a state change in the Modeler when the voltage returns to a fully charged state. At this time the Reasoner is again notified that a change in a state variable has occurred.
12. The Reasoner then notes the voltage has been restored to a fully charged level and marks the goal as accomplished.

4.3. ACT Operational Scenario

Figure 3 illustrates an operational scenario involving a possible ACT agent community for a nanosatellite constellation. It is based on the idea of a ground-based community of proxy agents, each representing a spacecraft in the nanosatellite constellation, which provide for autonomous operations of the constellation. Future scenarios will depict the migration of this community of proxy agents to the spacecraft for an evaluation of space-based autonomy concepts [5].

In this scenario there are several nanosatellites in orbit collecting magnetosphere data. The Mission Control Center (MCC) makes contact with selected spacecraft according to its planned schedule when the spacecraft come into view.

The agents that make up the MCC are:

- Mission Manager Agent (MMA): coordinates the agent community in the MCC, manages mission goals and coordinates with the Contact Manager Agent.
- Contact Manager Agent (CMA): coordinates ground station activities, communicates with the spacecraft, sends and receives data, commands, and telemetry.
- User Interface: interfaces with the user to get commands for the spacecraft and sends data to be displayed.
- MCC Planning/Scheduling Agent: plans and schedules contacts with the spacecraft via external planner/scheduler.
- Spacecraft Proxy Agents: keeps track of spacecraft status, health and safety, etc. The proxies notify the Mission Manager Agent when anomalies occur that need handling.

Each of the above agents registers with the GCC manager agent. The GCC manager agent notifies them when a contact is approaching for their spacecraft, if whether another agent is going to be added to the community, and how to contact another agent.

The following is a spacecraft contact scenario that illustrates how the agents work with the GCC manager agent:

- Agents register with the GCC Manager Agent at system startup.
- GCC Planner/Scheduler Agent communicates with the spacecraft Proxy Agents to get view data. It then creates a contact schedule for all orbiting spacecraft.
- GCC Manager Agent receives the schedule from the GCC Planner/Scheduler Agent and gives details of the next contact to the Contact Manager Agent: when and with which spacecraft.
- The Contact Manager Agent contacts the spacecraft at the appropriate time and downloads the telemetry and

sends it to the appropriate spacecraft Proxy Agent for processing.

- The spacecraft Proxy Agent processes the telemetry data, updates the spacecraft's status, evaluates any problems, warnings, etc., and sends any commands that need to be uploaded to the Contact Manager.
- If a Proxy Agent determines a problem exists and an extended or extra contact is needed, it sends a message is sent to the GCC Planner/Scheduler Agent, which will re-plan its contact schedule and redistribute it to the GCC Manager.
- The Contact Manager downloads data and uploads any commands from and to the spacecraft as instructed by the spacecraft Proxy Agent. The Contact Manager agent ends the contact when scheduled.

An example of a typical contact with a satellite would be:

- The Contact Manager Agent (CMA) receives an acquisition of signal (AOS) from a spacecraft. The MCC is now in contact with the spacecraft.
- The CMA requests the spacecraft to start downloading its telemetry data and sends received data to a spacecraft proxy agent.
- The proxy agent updates the state of its spacecraft model from the telemetry received. If a problem exists, the Mission Manager Agent is contacted and appropriate action (if any) is planned by the system.
- The Contact Manager Agent analyzes the downloaded telemetry data. If there is a problem, the CMA may alter the current contact schedule to deal with the problem.
- The CMA executes the contact schedule to download data, delete data, or save data for a future pass.
- The Mission Manager Agent ends contact.

4.4. Autonomic Properties of ACT

The various operational scenarios of ACT exhibit at least three types of activities that contribute to its autonomic functionality. The autonomic functionalities exhibited are: self-configuring (adaptation to changing environment), self-optimizing (steps to maximize utilization), self-healing (ability to recover from anomalies) and self-protecting (protect against failures). The following further discusses these autonomic properties of ACT.

4.4.1. ACT self-configuration. As an example of this property, when ACT detects, from analysis of downloaded telemetry, that there is a problem, the Contact Manager alters the current satellite contact schedule to enable the problem to be addressed. What is being reconfigured, in this case, is the subset of the spacecraft environment that is currently planned for contact.

4.4.2. ACT self-optimization. As an example of this property consider what happens when a Proxy Agent determines that a problem exists with its associated spacecraft. When this situation arises a replanning/rescheduling activity occurs. This is done to optimize the behavior of the entire ACT system.

4.4.3. ACT self-healing. As an example of this again consider what happens when a Proxy Agent detects a problem with its associated spacecraft. Following a diagnosis of the problem (which may involve access to the human component of the ACT) corrective actions, in the form of commands, are generated and made ready for transmission to the affected spacecraft. This problem-diagnosis/corrective-action cycle is a major part of ACT's self-healing capability.

It should be noted that the three autonomic features discussed above all stem from ACT's determination that a problem has occurred. In attending to the problem, ACT reconfigures, tries to optimize its operations and proceeds to diagnose and solve the identified problem.

4.4.4 ACT self-protection. ACT is self-protecting in the sense that it constantly monitors the spacecraft systems and modifies its operations if a parameter ranges outside its normal bounds. An example of self-protection is given in the above example of dataflow between components of the architecture. In this example, the battery is discharging and if nothing is done the spacecraft will lose power and become inoperable. ACT then takes the necessary actions to recharge the battery (turning towards the sun, for example). In addition, it also has self protection through validation of system commands to insure that command sequences executed will not harm the spacecraft or put it in a position where it could be harmed.

5. Conclusion

The LOGOS and ACT agent architectures and implementation were developed to demonstrate autonomous ground and space operations to lower costs of those operations and to provide technology for future missions requiring lengthy times between contacts with the ground. To accomplish these goals, autonomic properties of the systems were developed. Without these properties there would constantly have to be a human in the loop which would increase costs and limit the scope of future missions. This not only is true of NASA missions, but also would be true of other real-time mission critical systems in the commercial world.

The self-healing, self-configuring, self-optimizing and self-protecting properties of both LOGOS and ACT make them autonomic systems. It is also noted that in the spacecraft domain all of these properties are closely tied together. The self-healing actions can be the result of self-protecting actions. The self healing actions then may cause self-configuring commands to occur, which in turn may trigger self-optimizion.

The LOGOS and ACT architectures provide for a flexible implementation of a wide range of intelligent and autonomic agents. The ACT architecture allows for easy removal of unneeded components for reactive agents and the inclusion of the necessary components to implement intelligent and autonomic agents. It is also flexible so that additional unforeseen components can be added without affecting previous components.

The ultimate goal of our work is to be able to transition proven agent and autonomic technology into operational NASA systems. The implementation of the scenarios discussed above (and others under development) will provide an opportunity to exercise, evaluate and refine the capabilities supported by the agent architectures. It will also provide an opportunity for space mission designers and developers to "see" agent and autonomic technology in action and their resulting benefits. This will enable them to make a better determination of the role that this technology can play in their missions.

6. References

[1] Ferber, J. Multi-Agent Systems, An Introduction to Distributed Artificial Intelligence. Addison-Wesley. 1999.

[2] Foundation for Intelligent Physical Agents (FIPA). FIPA Specification Part 2: Agent Communication Language. Geneva, Switzerland. November 28, 1997.

[3] LOGOS Overview, Design and Specification Documents. http://agents.gsfc.nasa.gov/products.html.

[4] LOGOS System Overview Document. http://agents.gsfc.nasa.gov/documents/code588/LOGOS.stuff/logosoverview.pdf

[5] Rouff, C., and Truszkowski, W. A Process for Introducing Agent Technology into Space Missions. IEEE Aerospace Conference. March 11–16, 2001.

[6] Truszkowski, W., and Hallock, L. Agent Technology from a NASA Perspective. CIA-99, Third International Workshop on Cooperative Information Agents, Uppsala, Sweden, 31 July – 2 August 1999, Springer-Verlag.

[7] Truszkowski, W. and Rouff, C. An Overview of the NASA LOGOS and ACT Agent Communities. 5th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2001). July 22-25, Orlando, Florida.

[8] Wooldridge, M. Intelligent Agents. In Multiagent Systems. Edited by Gerhard Weiss. MIT Press. 1999

[9] Joseph, J. and Fellenstein, C. Grid Computing. IBM Press 2004.